RELATÓRIO TÉCNICO

/04

Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes

Gladys Machado Pereira Santos Lima
Guilherme Horta Travassos
ght@cos.ufrj.br

Programa de Engenharia de Sistemas e Computação
COPPE / UFRJ

Rio de Janeiro, 2004

CONTEÚDO

Resumo: Um importante problema quando realizando teste de integração em software orientado a objetos é decidir a ordem de integração das classes. Conceitos como encapsulamento, herança e polimorfismo adicionam complexidade aos testes, fazendo com que eventualmente dependências entre classes precisem ser quebradas. Além disto, devido aos problemas associados com a integração *big-bang,* as classes precisam ser integradas uma de cada vez ou, em alguns casos, em pequenos *clusters* [Briand, 2002b], tornando o processo de tomada de decisão sobre a ordem de integração a ser utilizada. O propósito deste estudo é identificar heurísticas que possam ser aplicadas aos diagramas de classes UML, de modo a estabelecer uma ordem de prioridade das classes para o teste de integração, utilizando o número de *stubs* de testes a serem implementados como métrica do esforço requerido.

Abstract: An important problem when integrating and testing object-oriented software is to decide the classes' integration order. Concepts such as encapsulation, inheritance and polymorphism had added complexity to testing, mainly for those situations when we need to break down dependences among classes. Due to the usual problems associated with the big-bang integration, classes need to be integrated one by one or, in some cases, in small clusters [Briand, 2002b], making this problem harder. This study aims to identify heuristics that can be applied to UML class diagrams, in order to establish the order that integration testing should be accomplished. Such approach uses the number of stubs as a metric to measure testing required effort.

Palavras-chave: Teste de Integração, Qualidade de Software, Engenharia de Software Orientada a Objetos, Experimentação.

1. Introdução

A introdução do paradigma orientado a objetos (OO) no contexto de desenvolvimento de software provocou mudanças significativas na forma como os produtos de software são criados e mantidos. As mudanças foram motivadas pela nova perspectiva adotada pelo paradigma OO (ênfase nos objetos), em oposição ao paradigma procedural, que utiliza uma abordagem focada na funcionalidade e no fluxo de informação dos sistemas.

Em conseqüência dessa nova visão para o desenvolvimento de software, muitas áreas tiveram que ser revisadas, pois teorias, práticas, modelos e métricas que eram adequados para software convencionais não podem ser aplicados de forma irrestrita quando se desenvolve software OO. Uma dessas áreas é o teste de software, no qual esse trabalho encontra-se inserido.

Segundo Vieira (1998), os principais fatores que diferenciam testes em software OO de testes em software convencionais são: encapsulamento; classes abstratas; herança; e

polimorfismo. Estes fatores adicionam complexidade que não existe quando se desenvolve software utilizando o paradigma convencional.

Embora muita pesquisa esteja sendo realizada em testes de software orientado a objetos, a relativa imaturidade desse paradigma é um fator que impacta a proposição e adoção de metodologias consistentes e reais para testar os produtos, pois ainda não se conhecem com exatidão todas as possibilidades e as limitações quando se utiliza esse paradigma.

O teste de integração constitui-se em uma atividade de se descobrir erros associados às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto [Maldonado in Rocha, 2001]. No entanto, de modo geral, as atividades que tem como objetivo garantir ou fornecer algum indicativo da qualidade de um produto, quando realizadas manualmente, tendem a ser sistematicamente negligenciadas.

A partir desta motivação, o objetivo deste estudo é identificar heurísticas que possam ser aplicadas aos diagramas de classes, de modo a estabelecer uma ordem de prioridade das classes para teste de integração.

2. Testes de Integração

A atividade de teste de software é dividida em fases, tanto na abordagem de desenvolvimento de software procedimental quanto na abordagem orientada a objeto e desenvolvida de forma incremental e complementar, segundo Maldonado [Rocha, 2001]. Basicamente, há três fases de testes: de unidade, de integração e de sistema.

No contexto da orientação a objetos, um componente pode ser entendido como uma unidade de teste básica, podendo corresponder a uma classe num sistema ou um método específico de uma classe. Um componente pode ser entendido como um sistema de componentes [Binder, 2000].

Os testes de unidade são testes de componentes individuais. Os testes do sistema resultam dos testes da união dos componentes. Os testes de integração são os testes das interações entre componentes, ou seja, é o caminho no qual o teste é conduzido para integrar componentes no sistema. Segundo Pfleeger (2004), o teste de integração é o processo de verificar se os componentes do sistema, juntos, trabalham conforme descrito nas especificações do sistema e do projeto do programa.

Uma estratégia de integração deve responder a três questões: quais componentes são foco dos testes de integração, em que seqüência as interfaces de componentes deverão ser exercitadas e qual técnica de teste será empregada para exercitar a interface? Teste de

integração é uma busca por defeitos que causam as falhas entre componentes [Binder, 2000].

2.1. *Stubs*

Os *stubs* são pedaços de software que devem ser construídos para simular partes de software que ainda não foram desenvolvidas ou ainda não foram testadas, mas necessários para testar as classes dependentes deles. Um s*tub* é a implementação parcial de um componente [Binder, 2000]. Um componente usado como fachada para simular o comportamento de um componente real [Beizer, 1984]. O teste de um componente A, que chama um componente B, mas B ainda não foi testado, implica na substituição de B por um componente chamado *stub*.

Um *stub específico* é escrito para simular o comportamento de B em relação ao componente A. Um *stub realístico* é escrito para simular o comportamento do componente B em qualquer caminho de teste [Le Traon *et al*, 2000]. *Stub realísticos* são confiáveis, apesar de se tornarem obsoletos, mas são implementações de componentes que precisam ter seu *stub* (*stubbed component ou stubbed class*).

2.2. Esforço de teste

O *stub* (pseudocontrolado) representa despesa indireta. É software que precisa ser escrito, mas que não é entregue com o produto final do software. Se os *stubs* são mantidos bem simples, as despesas indiretas reais são relativamente baixas. [Pressman, 2000].

O esforço de teste pode ser medido pelo número de *stubs* que precisam ser escritos conforme a estratégia de integração. O número de *stubs* é calculado dependendo do tipo que foi escrito. Se um *stub realístico* é usado, o esforço é 1. Se dois *stubs específicos* de um mesmo componente são escritos para testar outros dois componentes, então, o esforço é contabilizado em 2 [Le Traon *et al*, 2000].

2.3. Análise de Dependência

Segundo Binder (2000), os componentes, tipicamente, dependem uns dos outros de diversas maneiras. As dependências são necessárias para implementar colaborações e conseguir a separação de alguns interesses. Algumas dependências são acidentais ou efeitos colaterais de uma implementação, linguagem ou ambiente. As dependências no escopo das classes e *clusters* resultam de alguns mecanismos como: composição e agregação, herança, variáveis globais, uso de objetos como parâmetros de mensagens, entre outros.

De maneira semelhante, as dependências ocorrem entre componentes no escopo de um sistema. Algumas abordagens usam a análise de dependências para apoiar o teste de integração *bottom-up*. As dependências explícitas entre componentes correspondem às interfaces que necessitam serem exercitadas pelos testes de integração adequados [Binder, 2000].

3. Descrição do Problema

Tentar integrar a maioria ou todos os componentes no sistema ao mesmo tempo é normalmente problemático, pois alguns problemas de interface poderiam escapar da detecção [Binder, 2000].

Desde os primórdios de 1970, o teste de integração incremental baseado em estruturas físicas é empregado como uma técnica efetiva: adicionar componentes ao longo do tempo e, então, testar sua interoperabilidade. Por exemplo: o caminho de chamada dos módulos ou as comunicações entre processos serve como base para incremento dos módulos a serem testados em desenvolvimentos procedurais.

O desenvolvimento orientado a objetos (OO) trabalha com o encapsulamento das estruturas de informações e seus comportamentos, portanto, os programas OO não são executados de maneira seqüencial e seus componentes podem ser combinados de maneira arbitrária. O estado de um objeto não é determinado apenas por seus atributos, mas também pelas associações com outros objetos.

A dificuldade do entendimento da execução do programa OO traz um novo desafio, pois as estratégias tradicionais de teste na são capazes de lidar totalmente com as questões da orientação a objetos. A necessidade por testes é muito maior, uma vez que os erros podem “se esconder” pelos níveis de hierarquia e estarem relacionados ao estado de um objeto ao invés de uma seqüência de passos específica.

Alguns problemas identificados:

- *Classes abstratas*. Não é possível testá-las diretamente, pois não são instanciadas.
- *Herança*. Pequenas mudanças sintáticas podem acarretar grandes conseqüências semânticas, ou ainda podem resultar em estados e caminhos não tão óbvios a partir do código.
- *Polimorfismo*. Muitas vezes não é possível dizer o código que será executado quando da utilização de uma função polimórfica.

Segundo Pfleeger (2004), os objetos tendem a serem pequenos, e a complexidade que pode, geralmente, residir no componente, é freqüentemente transferida para a interface

entre componentes. Essa diferença quer dizer que o teste de unidades é menos difícil, ao passo que o teste de integração deve ser muito mais extensivo.

Os testes de unidade e de integração devem ser combinados para que se possa testar efetivamente uma implementação. Segundo Graham (1993) *in* Vieira (1998), duas estratégias devem ser aplicadas para sua realização:

- Para cada classe que é testada em uma hierarquia de classes, a classe de nível mais alto deve ser testada antes das classes de nível inferior. Isto envolve um teste "*Top-down*".
- Para *clusters* de classes que interagem entre si, os testes devem ser realizados em objetos que podem ser instanciados antes; posteriormente, os testes serão feitos em objetos que serão instanciados mais tarde, sendo esta uma abordagem "*Bottom-up*".

Devido aos problemas associados com a integração *big-bang,* as classes precisam ser integradas uma de cada vez ou, em alguns casos, em pequenos *clusters* [Briand, 2002b]. Dentro deste contexto, o problema pode ser descrito em como identificar uma ordem de prioridade das classes para testes de integração, ou seja, estabelecer critérios de precedência para verificar o funcionamento conjunto das classes.

O propósito deste estudo seria verificar, com base na semântica definida pelo paradigma orientado a objetos e representado nos modelos UML (*Unified Modeling Language*), quais as características determinantes para priorizar as classes a serem testadas antes, de modo a realizar satisfatoriamente o teste de integração.

Segundo Briand (2002b), um critério de avaliação para comparar ordens é o esforço de construção de *stubs* (esforço de teste) requeridos para integrar classes conforme a ordem específica. Se uma ordem de integração de classes necessita de outras classes que ainda não foram integradas, faz-se necessário o desenvolvimento de *stubs* para continuidade dos testes.

4. Trabalhos Relacionados

Alguns trabalhos relacionados foram encontrados na busca de uma solução (estratégias ou algoritmos) para determinar a ordem de integração das classes a partir do diagrama das classes. O objetivo destas abordagens é minimizar o número de *stubs* a serem produzidos durante os testes de integração das classes, com a finalidade de reduzir o fator de custo dos testes. Algumas destas abordagens empregam a análise das dependências para quebrar alguns ciclos de dependências e, então, criar os respectivos *stubs*.

4.1. Kung

Segundo Briand (2003), um dos primeiros trabalhos relacionados foi proposto por Kung *et al* [Kung *et al*, 1995 *in* Briand, 2003]. Este trabalho baseava-se no fato de existir ou não ciclos de dependências entre classes.

Caso não existissem os ciclos, então, definir a ordem de integração seria equivalente a executar uma ordenação topológica de classes baseada no grafo de dependências. Se existissem ciclos de dependência, a estratégia proposta consiste em identificar os componentes fortemente conectados (*strongly connected components* – SCC) e remover as associações até não existir mais os ciclos.

Entretanto, quando existe mais de uma associação candidata para interromper o ciclo, a escolha ocorre de forma aleatória. Ciclos de dependências são comuns em sistemas reais e a solução nestes casos pode envolver uma complexidade na solução dos ciclos de dependência.

Outras soluções que antecedem o problema são baseadas no princípio de quebra de algumas dependências para obter dependências acíclicas entre classes. No contexto deste estudo, a quebra da dependência implica que a classe destino deverá ter um *stub* quando na integração e teste da classe origem. A seguir são apresentadas algumas destas abordagens.

4.2. Tai e Daniels

Segundo Briand, Tai e Daniels propuseram uma estratégia que associava cada classe do diagrama de classes a números de nível: maior e menor. Estes números são usados para planejar uma ordem de integração. O número de nível maior é designado conforme as dependências de herança e de agregação, que não são quebráveis, ignorando as dependências da associação. O número de nível menor é designado baseado somente nas dependências de associação, que são quebráveis, dentro de cada nível principal. Nos níveis menores, os ciclos podem aparecer e devem ser quebrados a fim de aplicar uma classificação topológica.

Uma função peso é calculada para cada associação em cada nível principal como sendo o número de dependências de entrada de cada nó origem mais o número de dependências de saída do nó destino. O raciocínio usado é que quanto maior o peso, a quebra de dependências quebrará maior número de ciclos.

As associações que cruzam os níveis maiores no sentido da fonte para o destino, ascendentemente, ou seja, do número de nível menor para o maior. Desta forma, originando os *stubs* necessários. Os cruzamentos descendentes não são necessários quebrar, pois a classe destino já foi testada.

Nesta abordagem, somente as associações são quebradas, baseadas no argumento de que se outros tipos de dependências fossem quebrados mais complexos seriam os *stubs*.

4.3. Le Traon *et al*

Le Traon *et al* (2000) usam uma estratégia diferente para tratar com os ciclos de dependência. Essa abordagem identifica os componentes fortemente acoplados (SCC) por meio de algoritmos de busca em profundidade.

Os autores propõem uma estratégia baseada nos algoritmos de pesquisa em grafos (TDG – Grafos de Dependência de Teste) que reconhecem os SCC. O algoritmo é adaptado tal que cada corte no grafo é nomeado de acordo com um esquema da classificação. Um corte, o qual é usado para quebrar ciclos, é denominado como ramo de dependência. Um ramo de dependência é definido por uma raiz (classe) e seus ancestrais, isto é, uma raiz que possua classes como folhas em profundidade, ou seja, classes que dependam da raiz, diretamente ou indiretamente.

Le Traon *et al* quebra os ciclos de dependência removendo as dependências de maior peso que entram na classe, para o SCC considerado. O peso é definido diferentemente do de Tai e Daniels: é a soma de dependências que entram mais as que partem do ramo para a dada classe dentro do SCC em consideração. Resumindo, a noção de peso é definida em classes e focaliza especificamente na noção das dependências do ramo (que capturam alguns dos ciclos em que a classe é envolvida). Para cada SCC não trivial (com mais de uma raiz), o procedimento é chamado recursivamente.

Desde que o peso é computado de acordo com as dependências do ramo e estas dependem da construção da árvore de busca em profundidade, o peso depende da raiz da qual começasse o algoritmo de busca.

O algoritmo proposto pelos autores não é determinístico, pois depende de algumas decisões arbitrárias. Existem dois níveis de não determinismo. Primeiro, o resultado depende da classe inicial, por onde começa o algoritmo de busca. Segundo, o algoritmo não especifica o que fazer quando as classes mostram o mesmo peso, ou seja, dependendo do grafo pode haver um número diferente de *stubs*.

4.4. Briand *et al*

Uma outra abordagem é proposta por Briand *et al*. Esta abordagem também identifica os componentes fortemente conectados (SCCs) por meio de algoritmo baseado em grafos, mas difere das anteriores na maneira de quebrar os ciclos (isto é, não removendo relacionamentos de herança ou agregação).

Esta estratégia assemelha-se a de Le Traon *et al*., baseada em chamadas recursivas ao algoritmo de busca em profundidade. Entretanto, exibe também uma diferença importante, similar a estratégia de Tai e Daniels, usando a definição do peso para caracterizar as associações, computando uma estimativa do número dos ciclos em que a associação é envolvida em um SCC (componentes fortemente conectados).

Como na abordagem de Le Traon *et al*, Briand *et al* identificam os SCCs recursivamente usando o algoritmo de busca. Em cada etapa, isto é, dentro de cada SCC não trivial, calcula-se o peso de cada dependência da associação, usando uma versão modificada da definição de Tai e Daniels e, então, quebra-se a dependência da associação com maior peso.

Ao contrário da abordagem de Le Traon *et al*, esta estratégia não tem o primeiro nível não determinístico, desde que o algoritmo para computar SCCs não tenha nenhum efeito no peso. Nenhuma classificação da dependência é usada como ramo. Além disso, a respeito do segundo nível do não determinismo (escolhas alternativas para pesos iguais), ao contrário da definição de peso de Le Traon *et al*, todas as opções de escolha conduzirão ao mesmo número de *stubs* específicos. Isto é devido ao fato do critério conduzir à remoção de uma e a somente uma associação, em comparação ao critério usado por Le Traon *et al*. que conduz à remoção de cada dependência entrante da classe selecionada.

Na definição de Tai e Daniels, o peso de cada associação é o número de bordas entrantes da classe da fonte mais o número de bordas que parte da classe do alvo. Na abordagem de Briand *et al*, multiplica-se o número de bordas entrantes pelo número de bordas que parte dentro do SCC sob a consideração. É uma estimativa do número mínimo dos ciclos em que a associação é envolvida, dentro de um SCC. O objetivo é suprimir, primeiramente, as associações que são envolvidas no número o maior dos ciclos, a fim minimizar o número dos *stubs*. Este número é usado como uma heurística, o número estimado mínimo, como definição do peso, para selecionar a associação seguinte para ser quebrada no SCC.

4.5. Outras abordagens

Uma abordagem não baseada em grafos, mas que usa algoritmos genéticos, também propõe uma solução, para o problema da ordenação de classes para testes de integração, por meio de uma técnica de otimização global baseada em heurísticas. Foi desenvolvida pela comunidade de Inteligência Artificial (IA), entretanto, não será apresentada aqui, pois o foco foi investigar técnicas baseadas em grafos [Briand, 2002b].

Outras técnicas não baseadas em diagrama de classes existem, mas não são discutidas neste trabalho, devido ao fato de não terem como objetivo minimizar o número de *stubs* de teste [Briand, 2003].

5. Estudos de Caso

Briand *et al* (2003) realizaram cinco estudos de caso, para analisar o uso das três técnicas (Tai e Daniels; Lê Traon *et al*; eBriand *et al*) de ordenação baseadas em grafos (ORD) para execução de testes de integração, com o propósito de caracterizar resultados, com respeito ao custo-eficiência *(nº de stubs específicos necessários)* e prover informações sobre a complexidade dos *stubs* em termos de métodos e atributos envolvidos, do ponto de vista do pesquisador, no contexto de ambientes de desenvolvimento orientado a objetos utilizando modelos representados em ORD, obtidos por engenharia reversa de cinco sistemas desenvolvidas em Java.

Os casos escolhidos foram:

- Caso 1 - Sistema ATM (*Automated Teller Machine)*;
- Caso 2 – Ant (parte do projeto Jakarta, **http://jakarta.apache.org**);
- Caso 3 – SPM (*Security Patrol Monitoring*);
- Caso 4 – BCEL (*Byte Code Engineering Library*); e
- Caso 5 – DNS (implementação do *Domain Name System*).

Alguns detalhes dos sistemas são apresentados na Tabela 1.

Informação	ATM	Ant	SPM	BCEL	DNS
Classes	21	25	19	45	61
Usos (Clientes)	39	54	24	18	211
Associações e agregações	9	16	34	226	23
Composições	15	2	10	4	12
Heranças	4	11	4	46	30
Linhas de Código	1390	4093	1198	3033	6710

Tabela 1. [Briand, 2003]

Os cinco estudos de caso empregaram diagramas de relacionamento entre objetos - ORD (Object Relation Diagram), identificando classes e relacionamentos, obtidos por meio de engenharia reversa do código fontes dos referidos sistemas. Os resultados obtidos empregando cem vezes cada técnica foram sintetizados na Tabela 2.

Testes estatísticos foram realizados para análise dos valores, sendo empregado *Wilcoxon Rank-Sum*. Foi utilizado $\alpha = 0.01$.

Analisando os resultados pode-se ver que a técnica de Briand *et al* apresenta o melhor desempenho em relação às demais baseado no menor esforço de criação de *stubs*. Le Traon *et al* apresentou resultados melhores do que Tai e Daniels, entretanto, a distribuição de *stubs* mostra o aspecto não determinístico da técnica.

Sistemas	Briand *et al*	Le Traon *et al*	Tai e Daniels
ATM System			
No. de stubs	7	[7-22] 60%: 7	8
Atributos (custo)	[39-67] média: 54	[67-240] média:95	[52-80] média: 67
Métodos (custo)	[13-19] média: 16	[13-75] média: 25	21
Ant System			
No. de stubs	11	[13-22] 50%: 19	28
Atributos (custo)	[162-178]	[151-341]	[463-484]
Métodos (custo)	25	78	89
Métodos (custo)	26, 27, 28	57, 58, 72, 73	[37-52]
SPM System			
No. de stubs	17	27 (54%), 22 (46%)	20 (84%) 22 (16%)
Atributos (custo)	[146-151]	[251-284]	[203-262]
No. de stubs	70	67 68	128
Atributos (custo)	[114-120]	92 93	216 222
Métodos (custo)	72	324 329	182
DNS System			
No. de stubs	6	[6-16]	27
Atributos (custo)	19, 22, 28	[25-93]	61
Métodos (custo)	11	[59-96]	32

Tabela 2. Resumo dos resultados de [Briand, 2003].

6. Heurísticas Alternativas (versão inicial)

As heurísticas apresentadas por Briand *et al* foram realizadas com base em diagramas de relacionamento entre objetos, obtidos a partir da própria implementação das classes em estudo. Nesta proposta, diferentemente das anteriores, as heurísticas apresentadas visam o estudo das dependências entre classes em um nível de abstração mais elevado, empregando como modelo o diagrama de classes de um projeto, descrito pela UML (*Unified Modeling Language*), como base de entrada para todas as informações necessárias ao seu emprego.

A partir da aplicação de um conjunto de heurísticas que determinam critérios de precedência entre classes, poderá ser estabelecida uma lista ordenada das classes para

execução de testes de integração, anteriormente ao detalhamento de projeto e implementação do modelo em estudo.

6.1. Travassos e Oliveira

As heurísticas apresentadas a seguir foram retiradas de [Oliveira, 2003] e são fruto de observação e aplicação desta abordagem na academia e na indústria, discutidas no curso de Engenharia de Software Orientada a Objetos da COPPE / UFRJ por Travassos.

6.1.1. Critérios de Precedência

Os critérios de precedência foram definidos com o propósito de verificar, com base na semântica estabelecida pela UML, quais as características determinantes para classes serem testadas antes de outras, de modo a realizar satisfatoriamente os testes de integração, minimizando o número de *stubs* específicos a serem gerados. Os critérios de precedência são:

i) Herança

Considerando que as subclasses herdam as características e, principalmente, o comportamento das classes-base, garantir que a subclasse funcione de forma adequada significa garantir, primeiramente, que a superclasse tenha sido devidamente testada. Quando a superclasse for abstrata, deve-se testar primeiro a classe-filha que seja menos acoplada.

A análise das dependências em relação à classe-base propicia uma análise indireta das dependências da subclasse, na medida que a subclasse somente será testada após as classes das quais a classe-base depende terem sido testadas.

Uma classe-base concreta e devidamente testada não garante o bom funcionamento da subclasse, pois novas operações e atributos são adicionados, estendidos ou modificados, criando-se novos contextos não previstos na superclasse. No entanto, se o projeto OO é feito corretamente, tendo uma boa estrutura de herança, alguns dos testes usados na superclasse poderão ser usados na subclasse. Se a estrutura de herança não foi bem projetada, há que se fazer uma série de testes extras. Desta maneira, o custo de se usar uma herança mal feita é ter que retestar todo o código herdado, e isso, por vezes, pode ser impossível [Vieira, 1998].

ii)Assinatura dos métodos de uma classe

Se uma classe (cliente) utiliza serviços de outra classe (servidora), deve-se primeiro avaliar se estes serviços estão corretamente implementados, então, testa-se primeiramente a classe servidora e depois a classe cliente.

iii) Agregação

A modelagem de um relacionamento "*todo-parte*", na qual uma classe representa um item maior, formado por itens menores é chamada de agregação. A agregação divide-se em simples ou composta. Na agregação composta, o controle do *todo* sobre *as partes* é mais rígido, significando que *as partes* não podem existir além da existência do *todo* e que um objeto *parte* pode pertencer a apenas uma instância de *todo*, diferente da agregação simples, onde uma instância *parte* pode pertencer a mais de uma instância *todo*.

Neste contexto, considera-se a agregação simples e a composta com a mesma semântica, onde a classe *todo* depende dos serviços fornecidos pelas classes *partes*, então a classe *parte* na agregação terá precedência para teste de integração sobre a classe que representa o *todo*.

iv) Navegabilidade

A navegabilidade indica que uma classe torna-se atributo de outra. Considerando que associação denota dependência mútua, onde duas classes, que se encontram no mesmo nível hierárquico, colaboram para realizar um serviço, não há possibilidade de utilizar somente a semântica expressa pela associação para identificar precedência. Segundo Booch (1994), a menos que seja declarado explicitamente o contrário, uma associação implica navegação bidirecional.

Sendo assim, será utilizada a navegabilidade para definir critério de precedência quando a ligação entre as duas classes ocorrer através da associação. Nos casos que houver navegabilidade bidirecional serão analisadas as possibilidades de propiciar a escolha da classe a ser ordenada primeiramente pelo desenvolvedor, configurando um processo semi-automático.

v) Classes de Associação

Uma classe de associação surge a partir da necessidade de colaboração entre duas outras classes. Desta forma, antes de testar uma classe de associação é preciso garantir que a colaboração esteja funcionando adequadamente. Desta forma, as classes que deram

origem à classe de associação terão precedência de teste de integração sobre a classe derivada.

vi) Dependência

Uma dependência indica a ocorrência de um relacionamento semântico entre dois ou mais elementos do modelo onde uma classe cliente é dependente de alguns serviços da classe fornecedora, mas não tem uma dependência estrutural interna com esse fornecedor. No teste de integração, a classe cliente terá precedência para ser testada.

6.1.2. Fator de Integração e Fator de Integração Tardia

Para viabilizar a aplicação dos critérios de precedência e estabelecer a ordem de prioridade foram definidas duas propriedades: *Fator de Influência* (FI) e *Fator de Integração Tardia* (FIT).

i) Fator de Influência (FI)

O *fator de influência* (FI) de uma classe é um valor que quantifica a relação de precedência entre as classes, sendo, portanto, diretamente proporcional ao número de classes que precisam ser integradas posteriormente à classe em questão. Deve ser definido considerando os relacionamentos diretos da classe em questão. Quanto maior o número de classes que possuam relação de precedência com a classe sob análise, maior será seu *fator de influência*.

ii) Fator de Integração Tardia (FIT)

O *fator de integração tardia* (FIT) de uma classe expressa a relação que é estabelecida entre as classes após a definição do fator de influência e é obtido a partir da soma dos fatores de influência de todas as classes que têm precedência direta sobre a classe em questão. Quanto maior o *fator de integração tardia* de uma classe, mais tarde deve ser realizado o teste de integração para a classe em questão.

As propriedades fator de influência e fator de integração tardia são utilizadas para possibilitar a implementação da ordenação das classes por ordem de prioridade para teste de integração.

Exemplificando

A partir da combinação dos critérios de precedência, por meio dos fatores *de influência* (FI) e *de integração tardia* (FIT), que serão calculados para todas as classes

existentes do modelo, será possível estabelecer uma lista ordenada das classes para execução de testes de integração.

Em análise à Figura 1, observa-se que a classe *Escola* não tem precedência sobre nenhuma outra classe do modelo, sendo seu *fator de influência* igual a zero. *Departamento* tem precedência em relação à classe *Escola* (parte-todo), o que significa que seu *fator de influência* para a classe *Departamento* é igual a um, pois influencia apenas uma classe do modelo. Da mesma forma que a classe *Departamento*, a classe *Aluno* possui *fator de influência* igual a um, uma vez que influencia apenas uma classe do modelo, a classe *Escola*. Por outro lado, a classe *Curso* tem precedência sobre as classes *Departamento* e *Aluno*, seu *fator de influência*, então, é igual a dois.

A ordem de preenchimento da lista de classes surgirá, inicialmente, a partir da seleção daquelas classes que apresentarem *fatores de integração tardia* iguais a zero (Tabela 3), como calculado para a classe *Curso* do exemplo da Figura 1.

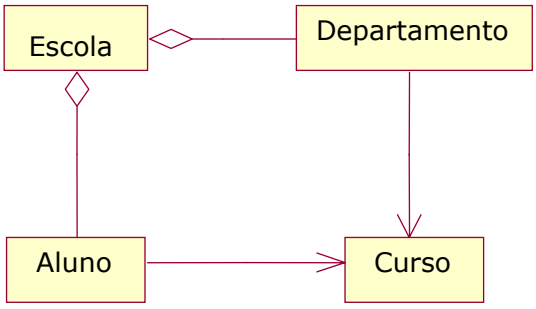


Figura 1.

CLASSE	FI	FIT
Escola	0	2
Departamento	1	2
Aluno	1	2
Curso	2	0

Tabela 3.

Para selecionar as próximas classes deverão ser recalculados os *fatores de integração tardia* das demais classes desprezando o *fator de influência* das classes anteriormente selecionadas (Figura 2) e, então, incluir na lista aquelas que apresentarem novo fator igual a zero (Tabela 4). Neste exemplo, o resultado da lista ordenada para teste de integração seria {*Curso*; *Departamento* ou *Aluno*; e *Escola*}.

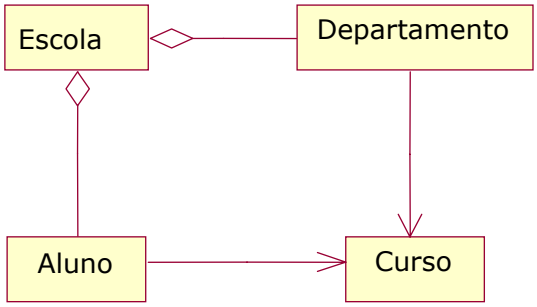


Figura 2.

CLASSE	FI	FIT
Escola	0	2
Departamento	1	0
Aluno	1	0
Curso	2	0

Tabela 4.

6.2. Evoluindo as Heurísticas

A aplicação das heurísticas em diagramas que não continham classes fortemente acopladas mostrou-se eficiente, entretanto, para diagramas contendo mais de uma classe com mesmo *fator de integração tardia*, representando ciclos de dependências entre classes, as heurísticas não resultaram um esforço de teste satisfatório, demandando um tratamento especial para situações de *deadlock*. Outras situações de melhoria das heurísticas foram observadas, em cursos, durante estudos de casos, bem como na observação direta de alguns aspectos, como a análise do *fator de influência* nulos, levando à necessidade de complementação das heurísticas, inicialmente sugeridas por Travassos e Oliveira (2003), as quais apresentamos a seguir.

6.2.1. Novo critério de precedência: a Cardinalidade

Concluímos que a cardinalidade, quando se analisa a navegabilidade bidirecional, também pode expressar um critério de precedência, com a seguinte definição:

vii) Cardinalidade

Um dos aspectos chaves em associações é a cardinalidade de uma associação, também chamada multiplicidade. A cardinalidade representa o número de objetos que participam em cada lado da associação, correspondendo à noção de obrigatório, opcional, um-para-muitos, muitos-para-muitos ou outras variações desta possibilidade, sendo especificada para cada extremidade da associação.

Será utilizada a cardinalidade para definir critério de precedência quando a cardinalidade representar a noção de opcionalidade (zero ou zero-para-muitos). Neste caso, a classe com cardinalidade opcional deverá ser testada após a outra classe da associação.

6.2.2. Análise do *Fator de Influência* Nulo

O resultado de valor nulo para o cálculo do fator de influência (FI) de alguma classe de um modelo é bastante significativo no processo de ordenação das classes. Expressa que a referida classe deverá ter seu teste de integração executado posteriormente a execução dos testes das demais classes com fatores de influência não nulos do modelo. Estas classes têm seu teste de integração totalmente dependente da integração das demais classes do modelo. Conceitualmente a classe com FI nulo representa a generalização de outras classes do modelo.

6.2.3. Inexistência de *Fatores de Integração Tardia* Nulos

Conforme exposto por Oliveira (2003), a idéia básica consiste em se buscar primeiramente para execução dos testes de integração as classes que se encontram com *fator de integração tardia* nulos. Entretanto, alguns modelos podem ser representados somente por classes com forte acoplamento, não existindo inicialmente classes com f*ator de integração tardia* nulos. Nestes casos, a seqüência ao teste de integração deve ser feita por meio das classes que possuam o menor FIT calculado.

6.2.4. Tratamento de *Deadlock*

Existem situações em que mais de uma classe do modelo apresenta o mesmo valor de FIT, significando que de alguma forma estas classes possuem uma dependência, existindo um ciclo. Nestes casos, a quebra da dependência implicará na implementação de um ou mais *stubs* para as classes destino daquela escolhida para dar continuidade ao teste de integração.

O tratamento dos ciclos será realizado por meio da integração de todas as classes que apresentarem o mesmo valor de *fator de integração tardia* (FIT), com a conseqüente geração de stubs específicos necessários, antes de subtrair o valor da influência destas classes dos valores do FIT das demais classes ainda não integradas, ou seja, antes de outra iteração para o cálculo do FIT.

Para estabelecer prioridade entre as classes com mesmo valor de FIT, deve-se respeitar alguns critérios:

- A classe selecionada deve gerar o menor número de *stubs* específicos necessários em comparação com o número de *stubs* para as outras classes com mesmo valor de FIT.
- No caso das classes necessitarem da implementação do mesmo número de *stubs*, deverá ser testada aquela que os *stubs* apresentarem a menor complexidade (medida pelo tamanho da classe, ou seja, pelo somatório do número de atributos e do número de métodos de cada *stubs,* [Lorenz, 1994]).
- No caso de alguma dessas classes ser testada diminuir o número de *stubs* necessários para testar as outras de mesmo FIT, indicando uma dependência interna, esta deverá ser testada primeiramente.
- Caso apresente alguma associação com navegabilidade obrigatória, a classe deverá ser testada primeira, preferencialmente.

6.3. Um processo para utilização das heurísticas

O processo foi modelado utilizando a abordagem de processos organizacionais apresentada em Villela *et al* (2001), sendo a notação apresentada na Tabela 5, permitindo uma representação gráfica do conhecimento e habilidades necessários à execução de uma dada atividade do processo.

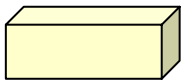
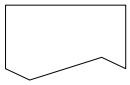
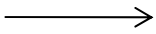
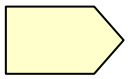
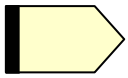
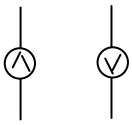
Objeto	Notação	Objeto	Notação
Processo		Documento	
Ator		Fluxo de Controle	
Atividade		Fluxo de Entrada/Saída	
Atividade Composta		Operação Lógica	 and or

Tabela 5. Entidade e Forma de Representação

O processo de aplicação das heurísticas é utilizado para gerar a lista ordenada das classes de um modelo, representado pelo diagrama de classes, como orientação da ordem necessária para execução dos testes de integração destes componentes, num paradigma orientado a objetos, com o objetivo de minimizar o esforço do teste, medido pelo número de *stubs* específicos produzidos. A seguir são listadas as atividades e sub-atividades do processo de aplicação das heurísticas proposto, que é dividido nos processos *Pesquisar Fator de Influência*; *Tratar Fator de Integração Tardia*; e *Tratar Classes Dependentes*, conforme observado na Figura 3.

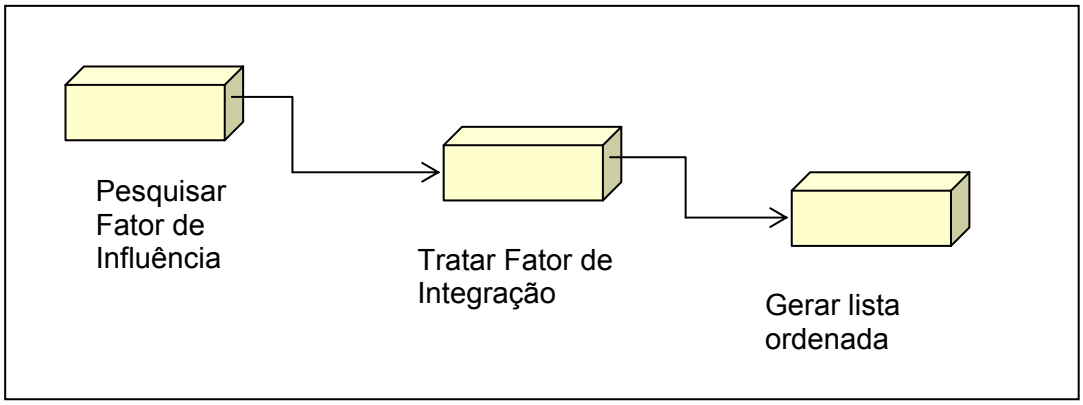


Figura 3. Processo de aplicação das heurísticas

6.3.1. Pesquisar Fator de Influência

O objetivo é calcular o valor do *fator de influência* (FI) para todas as classes existentes no modelo, aplicando os critérios de precedências estabelecidos, e separar aquelas classes com FI nulo, gerando uma lista de classes que terão sua ordenação estabelecida posteriormente.

Atividades:

- Aplicar critérios de precedência: identificar para cada classe aquelas que serão integradas posteriormente à classe em questão, seguindo os critérios estabelecidos.
- Atribuir FI: associar o número de classes integradas posteriormente a classe em questão ao valor do *fator de influência,* para cada classe do modelo.
- Separar classes totalmente dependentes: identificar e separar as classes com *fator de influência* nulo (FI=0), gerando as listas de classes totalmente dependentes (LCD) e de classes não ordenadas (LCNO).

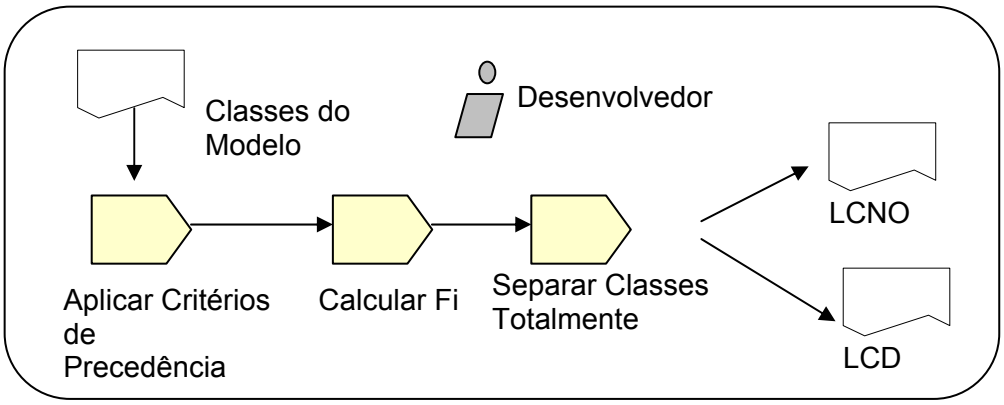


Figura 4. *Pesquisar Fator de Influência*

6.3.2. Tratar Fator de Integração Tardia

O objetivo é estabelecer uma ordem para integração das classes, calculando o *fator de integração tardia* (FIT) das classes a cada iteração necessária para buscar a classe folha (FIT = 0) ou aquelas com menor FIT, ordenadas segundo o critério de priorização destas.

Atividades:

- Identificar precedências: estabelecer para cada classe aquelas que possuem precedência de integração em relação a classe em questão.
- Calcular FIT: enquanto houver classes para serem ordenadas, calcular o valor do fator de integração tardia das classes.
- Buscar classes folha: a cada interação, identificar as classes com FIT nulos como aquelas a serem incluídas no final da lista de classes ordenadas.

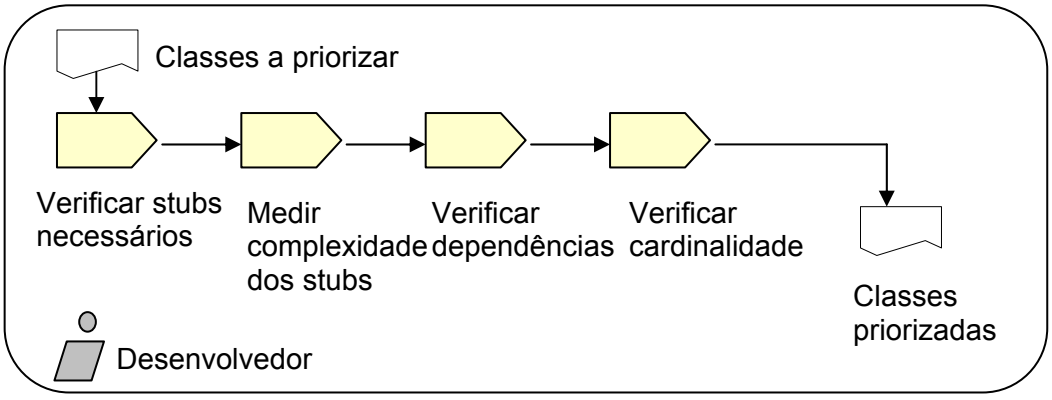


Figura 5. Detalhamento da atividade “Priorizar classes não folha”.

- Priorizar classes não folha: a cada interação, caso não existam classes com FIT nulos, identificar e estabelecer uma prioridade entre classes que possuam o mesmo valor de FIT, o menor a cada interação. Baseado no tratamento de *deadlock*, esta atividade possui as seguintes sub-atividades:
    - Verificar *stubs* necessários: estabelecer, quais e quantos, *stubs* específicos serão necessários para integrar cada classe não folha.
    - Medir complexidade dos *stubs*: no caso do número de *stubs* calculados para as classes na atividade anterior sejam iguais, deverá ser calculado o tamanho dessas classes como critério de prioridade da ordem.

- Identificar relação de dependências: no caso do tamanho calculado na atividade anterior sejam iguais, deverá ser identificada a possibilidade de alguma classe contribuir para integração de outras e utilizar para estabelecer o critério de prioridade.
- Verificar a cardinalidade das associações, caso exista alguma com opcionalidade, utilizar como critério de prioridade. Caso não seja possível, estabelecer uma ordem aleatória.

- Atualizar listas: incluir as classes selecionadas na lista ordenada de classes para o teste de integração (LCOTI) e retira-las da LCNO.
- Reduzir a influência das classes ordenadas: a medida que foram selecionadas classes folhas ou classes priorizadas com menor FIT, o valor dos respectivos FI deverão ser subtraídos para nova execução da atividade de "Calcular FIT".

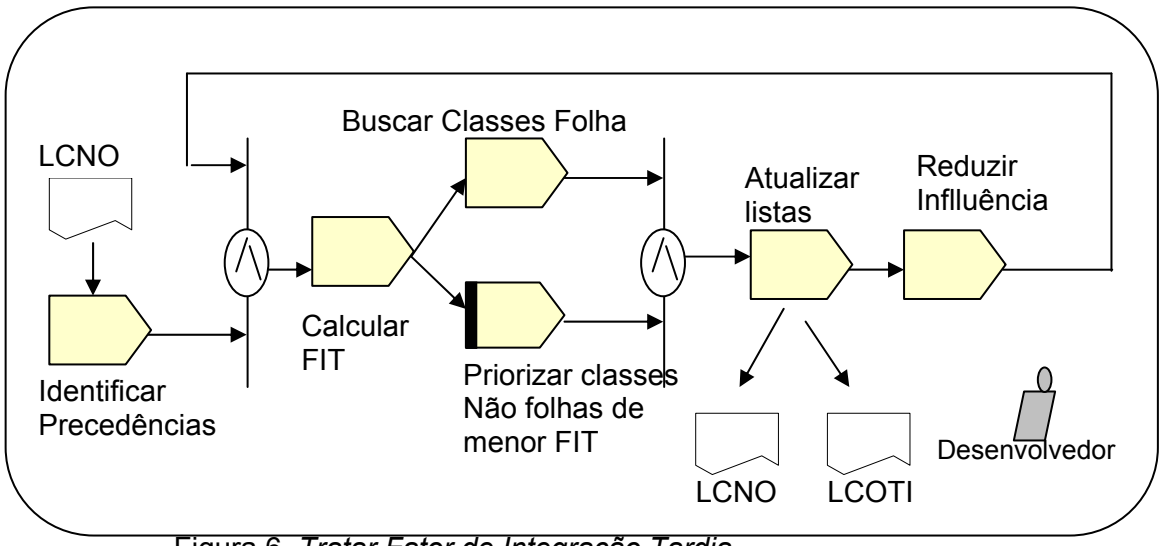


Figura 6. *Tratar Fator de Integração Tardia*

6.3.3. Gerar Lista Ordenada de Classes

O objetivo é inserir, após a ordenação estabelecida no processo anterior, as classes totalmente dependentes isoladas no processo de *Pesquisar Fator de Influência.*

Atividade:

- Inserir classes totalmente dependentes: adicionar ao final da lista previamente ordenada (LCOTI) a LCD originadas pela atividade "Separar classes totalmente dependentes" no processo de *Pesquisar Fator de Influência*.

6.4. Estudo de Caso: o modelo de Briand

Para ilustrar as técnicas estudadas e compará-las com sua própria proposta, Briand (2003) utilizou como exemplo o diagrama de relacionamento entre objetos (ORD) da Figura 7, onde: heranças são identificadas por ***I***; agregações por ***Ag***; associações por ***As;*** e as classes por letras maiúsculas.

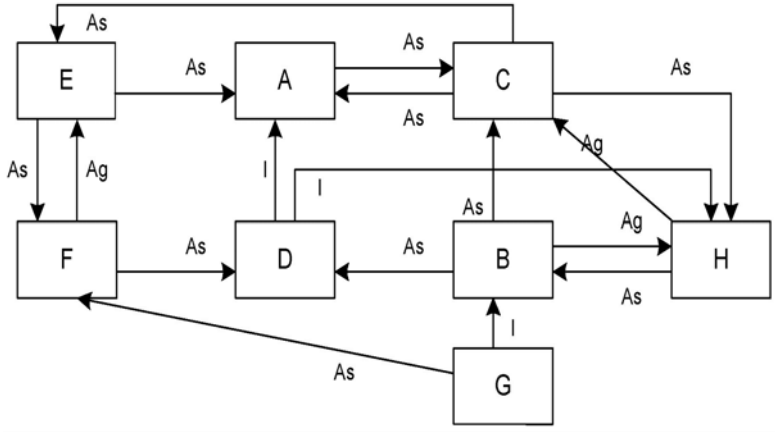


Figura 7. Exemplo extraído de [Briand, 2003].

A tabela 6 resume os resultados obtidos com a aplicação das estratégias relatadas por Briand (2003) sobre o modelo da Figura 7. Podemos observar que Le Traon *et al* apresentam melhores resultados do que Tai e Daniels. Como o resultado de Briand *et al* é determinístico, seu esforço de teste pode ser considerado melhor que o apresentado por Le Traon *et al*, pois este último somente apresenta resultado igual ao de Briand quando o vértice escolhido inicialmente for G.

Proposta	**Seqüência de Teste**	***Stubs* Específicos**
Tai e Daniels	{A, E, C, F, H, D, B, G}	Stub (C, A) Stub (F, E) Stub (H, C) Stub (D, F) e Stub (B, H)
Le Traon *et al* (começando com o nó G)	{A, H, D, E, F, C, B, G}	Stub (C,A) Stub (B, H) e Stub(C, H) Stub (F,E)
Briand *et al*	{A, E, C, H, D, F, B, G}	Stub (C,A) Stub (F,E) Stub (H, C) Stub (B, H)

Tabela 6.

Tendo por base os conceitos utilizados em [Briand, 2002a] para efetuar a relação entre um diagrama de classes e seu correspondente ORD, foi gerado o diagrama de classes da Figura 8 para demonstrar o emprego das heurísticas alternativas proposta neste trabalho, considerando que todas as classes podem ser instanciadas.

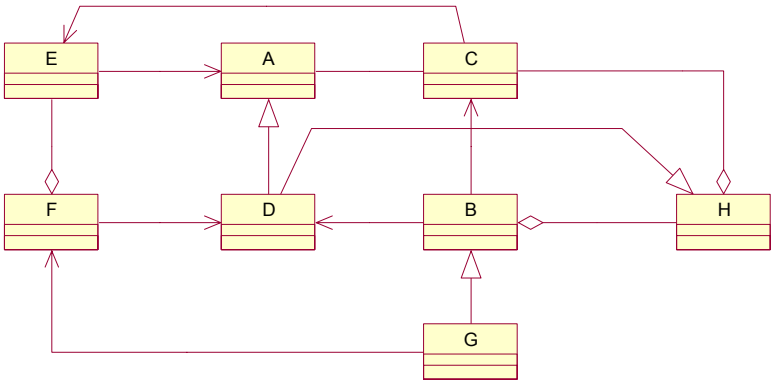


Figura 8. Diagrama de Classes correspondente a Figura 7.

Tabela 7.

Classes	FI
A	3
B	1
C	3
D	2
E	2
F	1
G	0
H	2

Ao aplicar o processo *Pesquisar Fator de Influência,* a lista de classes não ordenada seria composta por LCNO={A, B, C, D, E, F, G, H,} e obteremos os valores de *fatores de influência* apresentados na Tabela 7. A classe G por possuir *fator de influência* nulo será retirada da LCNO por ser totalmente dependente dos testes de integração das classes B e H, que por sua vez são dependentes das demais classes, sendo, então, LCD={G}.

Tabela 8.

Classes	FI	FIT 1ª. interação	FIT 2ª. interação	FIT 3ª. interação
A	3	**3**		
B	1	7	2	**0**
C	3	5	**0**	
D	2	5	**0**	
E	2	**3**		
F	1	4	2	**0**
H	2	**3**		

O processo *Tratar Fator de Integração Tardia* para nova lista LCNO={A, B, C, D, E, F, H} inicia-se com os valores mostrados pela Tabela 8. Observando a inexistência de *fatores de integração tardia* nulos,

as classes A, E e H serão priorizadas conforme a necessidade da dependência interna entre E e A, pois ambas necessitam de 1 *stub* específico. A lista de classes ordenadas para teste de integração será, neste momento, LCOTI={A, E, H}.

Ao "Reduzir a influência" de A, E e H, teremos que calcular o FIT, segunda interação, da LCNO={B, C, D, F}. A atividade de "Buscar classes folha" será executada e as classes C e D atualizam a LCOTI. Como os valores de FIT para as classes F e B na terceira interação são nulos, a LCOTI={A, E, H, D, C, F, B, G} é finalmente estabelecida com a inclusão da classe G ao final mesma, durante o processo de *Gerar Lista Ordenada de Classes*.

A utilização da seqüência de ordenação das classes para os testes de integração encontrada seguindo as heurísticas apresentadas neste trabalho, implicará na necessidade de implementação de dois *stubs* específicos: um *stub* de C para testar A e um outro *stub* de C para testar H, determinando um esforço de teste igual a dois, ou ainda, na implementação de um *stub* realístico de C para ambas as classes A e C. Comparando este resultado com o valor de quatro *stubs* para a ordem estabelecida por Briand *et al*, podemos observar que houve uma redução substancial do esforço de teste. Alie-se a isto, a facilidade de entendimento das abstrações representadas num diagrama de classe UML comparativamente ao manuseio de um ORD.

7. Estudo Experimental

Para avaliar de forma mais abrangente a aplicação das heurísticas em diferentes situações de projeto e tentar, minimamente, ampliar a abrangência dos resultados encontrados, evitando qualquer viés ou risco de aplicação em apenas um modelo, nos leva a identificar a necessidade de elaborar estudo experimental visando a caracterização destas heurísticas. A seguir é apresentado o planejamento deste estudo, conforme proposto por Wohlin *et al* (2000).

7.1. Objetivo do Estudo

Analisar *o uso de heurísticas para estabelecer a ordem de precedência de classes para testes de integração.*

Com o propósito de *caracterizar.*

Com respeito ao *custo-eficiência (nº de stubs específicos necessários).*

Do ponto de vista *do pesquisador.*

No contexto dos *mesmos sistemas nos quais foram testadas as técnicas anteriormente apresentadas por Lionel Briand (ATM, Ant, SPM, BCEL e DNS), ambientes de desenvolvimento orientado a objetos utilizando modelos representados em UML.*

7.1.1. Questões

Q1. O número de *stubs* específicos necessários para a ordem de precedência das classes estabelecida pelo emprego da heurística proposta foi menor do que o resultado apresentado pela técnica de Brind *et al*.?

M1. Número de *stubs* específicos para cada diagrama de classe aplicado.

Q2. O esforço necessário para aplicação dos testes de integração foi menor do que o resultado apresentado por Brian *et al*.?

M2. Número de atributos e métodos dos *stubs* necessários.

7.2. Planejamento

7.2.1. Definição das Hipóteses

Neste contexto, utilizaremos as seguintes definições:

G – quando utilizando as heurísticas apresentadas neste estudo.

B – quando utilizando as heurísticas apresentadas por Briand (2003).

S – número de stubs necessários identificados.

A – número de atributos necessários para os *stub* identificados.

M – número de métodos necessários para os *stub* identificados.

Hipótese Nula (H0): As heurísticas apresentadas não fornecem benefícios para os testes de integração baseados nos diagramas.

H0: S(G) > S(B) ∧ A(G) > A(B) ∧ M(G) > M(B)

Hipótese alternativa (H1): As heurísticas apresentadas empregam um número de *stubs* específicos necessários para aplicação dos testes de integração menor ou igual do que a heurística de Briand *et al*.

H1: S(G) =< S(B)

Hipótese alternativa (H2): As heurísticas apresentadas representam um esforço menor na criação dos *stubs* específicos necessários para aplicação dos testes de integração igual ou menor do que na criação dos *stubs* da heurística de Briand *et al*.

H2: A(G) =< A(B) ∨ M(G) =< M(B)

7.2.2. Descrição da Instrumentação

Será empregada a mesma ferramenta de testes estatísticos do estudo de caso anterior, *Wilcoxon Rank-Sum.*

7.2.3. Seleção do Contexto

O contexto pode ser caracterizado conforme quatro dimensões: o processo (on-line / off-line); os participantes; realidade (o problema real / modelado); e generalidade (específico / geral).

Este estudo supõe o processo off-line porque as heurísticas não estão sendo aplicadas durante um ciclo de desenvolvimento dos sistemas propostos. Os participantes deste estudo serão os diagramas de classes selecionados. Serão selecionados diagramas de sistemas em uso, portanto aplicados a problemas reais e não modelados. As heurísticas são comparadas às apresentadas por Briand para os mesmos modelos, então, o contexto possui um caráter específico.

7.2.4. Seleção dos Indivíduos

Os participantes deste estudo serão os diagramas de classes dos sistemas: ATM (*Automated Teller Machine)*; Ant (parte do projeto Jakarta, http://jakarta.apache.org); SPM (*Security Patrol Monitoring*); BCEL (*Byte Code Engineering Library*); e DNS (implementação do *Domain Name System*), ou seja, os mesmos do estudo apresentado por Briand para possibilitar a comparação dos resultados.

7.2.5. Variáveis

Variável independente:

Conjunto de heurísticas para determinar a ordem de precedência das classes para execução do teste de integração.

Variável dependente:

Custo da aplicação, medido pelo número de *stubs* específicos a serem criados para os testes.

7.2.6. Análise Qualitativa

Não será executada análise qualitativa devido a objetividade das métricas utilizadas.

7.2.7. Validade

Validade Interna: serão utilizados diagramas de classes de sistemas reais, portanto, assume-se que são representativos.

Validade de conclusão: como serão empregados os mesmos diagramas de classes dos sistemas do estudo de Briand, então, serão válidos para efeito de comparação.

Validade de construção: como serão utilizadas as novas heurísticas como novo tratamento a ser aplicado nos mesmos modelos da técnica a ser comparada, Briand *et a*l, o resultado a ser observado, o número de *stubs* específicos necessários, refletirá o comportamento a ser estudado.

Validade externa: serão empregadas as técnicas fora do ambiente de desenvolvimento de software, em diagramas de classes de sistemas prontos, portanto, não é possível generalizar os resultados obtidos para a indústria.

8. Conclusões e Trabalho Futuro

As heurísticas apresentadas neste trabalho precisam ser refinadas, para tal, devem ser aplicadas a diversos diagramas de classes, referentes preferencialmente a sistemas reais, onde outros passos diferentes aos descritos devem ser analisados, para sua efetiva inclusão na técnica em questão.

É fundamental a execução de um estudo de viabilidade e comparação das heurísticas apresentadas em relação ao trabalho de Briand *et al*, conforme proposto no Estudo Experimental detalhado anteriormente.

Devido ao processo manual de cálculo dos fatores de influência (FI) e integração tardia (FIT), para diagramas com grande número de classes e relacionamentos, ser dispendioso, deve-se preparar, anteriormente à aplicação do estudo experimental de caracterização das heurísticas, uma ferramenta para automatizar o processo de ordenação das classes para aplicação dos testes de integração. A possibilidade de automatizar, também, a geração da lista de *stubs* específicos necessários ao teste deverá ser estudada.

A melhoria nos testes pode reduzir os custos de desenvolvimento de software ou ainda melhorar o desempenho. Desta maneira, podemos pensar na ordem de integração das classes como guia na determinação da ordem de implementação das classes, o que poderá ajudar na redução do tempo requerido para o desenvolvimento e teste de sistemas.

Agradecimentos

Ao Hamilton Oliveira pelo trabalho inicial com as heurísticas para integração de classes. À Marinha do Brasil pela oportunidade de participação desta Oficial no Curso de Mestrado em Engenharia de Sistemas e Computação do Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia - COPPE / UFRJ.

Este trabalho está sendo realizado no contexto do projeto CNPq – 475407/2003-2.

Referências Bibliográficas

[Antoniol, 2002] Antoniol, G.; Briand, L.C.; Di Penta, M.; Labiche, Y.; **A case study using the round-trip strategy for state-based class testing**, Proceedings of the 13th International Symposium on Software Reliability Engineering, 12-15 Nov. 2002, Page(s): 269 –279

[Beizer, 1984] Beizer, B.; **Software System Testing and Quality Assurance**; Van Nostrand Reinhold Company Inc, 1984.

[Binder, 2000] Binder, R.V.; **Testing object-oriented systems: models, patterns, and tool;** Addison-Wesley, 2000.

[Booch, 2000] Booch, G., Rumbauch, J., Jacobson, I., **UML – Guia do Usuário**, Editora Campus, 2000.

[Briand, 2001] Briand, L.C.; Labiche, Y.; Yihong Wang; **Revisiting strategies for ordering class integration testing in the presence of dependency cycles**, ISSRE 2001. Proceedings of the. 12th International Symposium on Software Reliability Engineering, Nov. 2001, Page(s): 287 -296

[Briand, 2002a] Briand, L.C.; Labiche, Y.; Soccar, G.; **Automating impact analysis and regression test selection based on UML designs**, Software Maintenance, 2002. Proceedings. International Conference on, 3-6 Oct, 2002.

[Briand, 2002b] Briand, L.C.; Feng, J. ; Labiche, Y.; **Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders**, Carleton University, Technical Report SCE-02-03, Version 3, Oct, 2002.

[Briand, 2003] Briand, L.C.; Labiche, Y.; Yihong Wang; **An investigation of graph-based class integration test order strategies**, IEEE Transactions on Software Engineering, 0098-5589/03, Vol. 29, Issue: 7, July, 2003, Page(s): 594 -607

[Furlan, 1998] Furlan, J.D.; **Modelagem de Objetos através da UML, Makron Books**, São Paulo, 1998.

[Kitchenham, 2002] Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K.; Rosenberg, J.; **Preliminary guidelines for empirical research in software engineering**; Software Engineering, IEEE Transactions on , Volume: 28 Issue: 8 , Aug. 2002, Page(s): 721 –734.

[Le Traon , 2000] Lie Traon, Yves; Jéron, T.; Jézéquel, J.; e Morel, P., **Efficient Object-Oriented Integration and Regression Testing,** IEEE Transactions Reliability, Vol. 49, no. 1, Page(s): 12-25, 0018-9529/00, 2000.

[Lorenz, 1994] Lorenz, M., Kidd, J.; **Object-Oriented Metrics: A Pratical Guide**, Prentice Hall, USA, 1994.

[McGregor, 1994] McGregor, J.D.; Korsons, T.D., **Integrated Object-Oriented Testing and Development Processes,** Communications of the ACM, September 1994, vol. 37, no. 39, page(s): 59-77.

[Furlan, 1998] Furlan, J.D.; **Modelagem de Objetos através da UML, Makron Books**, São Paulo, 1998.

[Kitchenham, 2002] Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K.; Rosenberg, J.;

Preliminary guidelines for empirical research in software engineering; Software Engineering, IEEE Transactions on , Volume: 28 Issue: 8 , Aug. 2002, Page(s): 721 –734.

[Le Traon , 2000] Lie Traon, Yves; Jéron, T.; Jézéquel, J.; e Morel, P., **Efficient Object-Oriented Integration and Regression Testing,** IEEE Transactions Reliability, Vol. 49, no. 1, Page(s): 12-25, 0018-9529/00, 2000.

[Lorenz, 1994] Lorenz, M., Kidd, J.; **Object-Oriented Metrics: A Pratical Guide**, Prentice Hall, USA, 1994.

[McGregor, 1994] McGregor, J.D.; Korsons, T.D., **Integrated Object-Oriented Testing and Development Processes,** Communications of the ACM, September 1994, vol. 37, no. 39, page(s): 59-77.

[Oliveira, 2003] Oliveira, H.; **Construção de um componente genérico baseado em heurísticas para ordenação das classes em ordem de prioridade de teste de integração,** Estudo do Laboratório de Engenharia de Software, COPPE, UFRJ, 2003.

[Pfleeger, 2004] Pfleeger, S. L., **Engenharia de Software: Teoria e Prática***,* Prentice Hall, 2ª. Edição, 2004.

[Pressman, 2001] Pressman, R. S., **Engenharia de Software***,* Mc Graw Hill, 5ª. Edição, 2001.

[Richardson, 1989] Richardson, D.J.; Aha, S.L.; Osterweil, L.J., **Integrating Testing Techniques Through Process Programming,** University of California, ACM 089791-342-6/89/0012/0219, 1989.

[Rocha, 2000] Rocha, A. R. C., Maldonado, J. C., Weber, K. C., **Qualidade de Software: Teoria e Prática***,* Prentice Hall, 2001.

[Travassos, 2002] Travassos, G.H.; Gurov, D.; Amaral, E.A.G.G., **Introdução à Engenharia de Software Experimental**, Relatório Técnico ES-590/02-Abril, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ.

[Vieira, 1998] Vieira, M.E.R.; **Abordagem para Apoio ao Teste Baseado no Comportamento de Sistemas Orientados a Objetos,** Tese de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE, UFRJ, Rio de Janeiro, 1998.

[Villela, 2001] Villela, K., Santos, G., Bonfim, C. *et al*; **Knowledge Management in Software Development Environments,** 14th Internacional Conference Software & Systems Engeneering and their Aplications, Paris, Dezembro, 2001.

[Wohlin, 2000] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.; **Experimentation in Software Engineering: an introduction,** Kluwer Academic Publishers, USA, 2000.